Integer Programming Based Algorithms for Peg Solitaire Problems (Algorithm Engineering as a New Paradigm)

AUTHOR(S):

Kiyomi, Masashi; Matsui, Tomomi

数理解析研究所講究録
1185 巻 2001 年 100-108

Integer Programming Based Algorithms for Peg Solitaire Problems

Masashi KIYOMI[1], Tomomi MATSUI[1]
[1]Department of Mathematical Engineering and Information Physics,
Graduate School of Engineering, University of Tokyo,
7-3-1, Hongo, Bunkyo-ku, Tokyo 113-8656, Japan.
{masashi, tomomi}@misojiro.t.u-tokyo.ac.jp

Abstract: Peg solitaire is a classical one player game. In this paper, we dealt with the peg solitaire problem as an integer programming problem. We proposed algorithms based on the backtrack search method and relaxation methods for integer programming problem.

The algorithms first solve relaxed problems and get an upper bound of the number of jumps for each jump position. This upper bound saves much time at the next stage of backtrack searching. While solving the relaxed problems, we can prove many peg solitaire problems are infeasible. We proposed two types of backtrack searching, forward-only searching and forward-backward searching. Our algorithm can solve all the peg solitaire problem instances we tried and the total computational time is less than 20 minutes on an ordinary notebook personal computer.

Keywords: peg solitaire, integer programming, backtrack searching.

1 Introduction

Peg solitaire is a one player game using pegs and a board with some holes. In each configuration, each hole contains at most one peg (see Figures 1, 2). A peg solitaire game has two special configurations, the starting configuration and the finishing configuration. The aim of this game is to get the finishing configuration from the starting configuration by moving and removing pegs as follows.

> When there are (vertically or horizontally) consecutive three holes satisfying that first and second holes contain pegs and third hole is empty, the player can remove two pegs from the consecutive holes and place a peg in the empty hole (see Figure 3).

The move obeying the above rule is called a *jump*.

In this paper, we propose an algorithm for peg solitaire games based on integer programming problems. In Sections 2, 3, and 4, we consider a peg solitaire problem defined below, which is a natural extension of the ordinary peg solitaire game. In Section 2, we formulate the peg solitaire problem as an integer programming problem. In Section 3, we show a relation between the pagoda function defined by Berlekamp, Conway and Guy [3] and our integer programming problem. Section 4 proposes an integer programming problem which is useful for pruning the backtrack search described in Section 6. We report computational results of our algorithm in Section 5.

There are many types of peg solitaire boards and various pairs of starting and finishing configurations. The most famous peg solitaire board is that of Figure 1, which is called the English board.

A peg solitaire problem is defined by a peg solitaire board and a pair of starting and finishing configurations. If there exists a sequence of jumps which transforms the starting configuration into the finishing configuration, we say that the given peg solitaire problem is *feasible*, and the sequence of jumps is a *feasible sequence*. The peg solitaire problem finds a feasible sequence of jumps if it is feasible; and answers "infeasible", if the problem is not feasible.

In [8], Uehara and Iwata dealt with the generalized Hi-Q problems which are equivalent to the above peg solitaire problems and showed the NP-

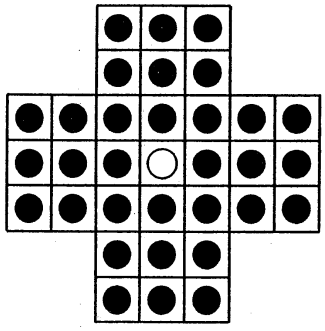

Figure 1: starting configuration example

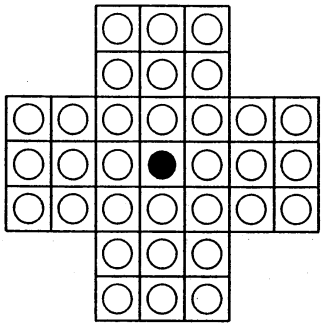

Figure 2: finishing configuration example

● implies a hole with a peg, and ○ implies a hole with no peg.

completeness. In the well-known book "Winning ways for Mathematical Plays [3]", Berlekamp, Conway and Guy discussed variations of problems related to peg solitaire problems. They showed the infeasibility of the peg solitaire problem "sending scout 5 paces out into desert" by using the pagoda function approach. In [7], Kanno proposed a linear programming based algorithm for finding a pagoda function which guarantees the infeasibility of a given peg solitaire problem, if it exists. Recently, Avis and Deza [1] formulated a peg solitaire problem as a combinatorial optimization problem and discussed the properties of the feasible region called "a solitaire cone".

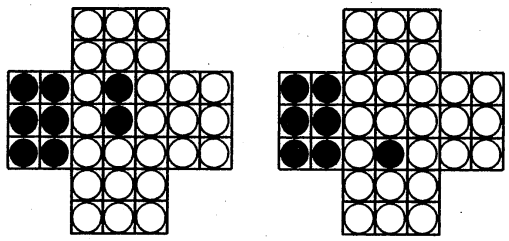

Figure 3: an example of a jump

2 Integer Programming

In this section, we formulate the peg solitaire problem as an integer programming problem.

We assume that all the holes on a given board are indexed by integer numbers $\{1, 2, \ldots, n\}$. The board of Figure 1 has 33 holes and so $n = 33$. We describe a state of certain configuration (pegs in the holes) by the n-dimensional 0-1 vector $\boldsymbol{p}$ satisfying that the ith element of $\boldsymbol{p}$ is 1 if and only if the hole i contains a peg. In the rest of this paper, we denote the starting configuration by $\boldsymbol{p}_s$ and the finishing configuration by $\boldsymbol{p}_f$.

Let J be the family of all the sequences of consecutive three holes on a given board. Each element in J corresponds to a certain jump and so we can denote a jump by a unit vector indexed by J. In the rest of this paper, we assume that all the elements in J are indexed by $\{1, 2, \ldots, m\}$. For example, the board of Figure 1 contains 76 sequences of consecutive three holes and so $m = 76$. Given a peg solitaire board, we define $n \times m$ matrix $A = (a_{ij})$, whose rows and columns are indexed by holes and jumps respectively, by

$$a_{ij} = \begin{cases} 1 & \text{(a peg on the hole } i \text{ is} \\ & \text{removed by the jump } j\text{)}, \\ -1 & \text{(a peg is placed on the hole } i \\ & \text{by the jump } j\text{)}, \\ 0 & \text{(otherwise)}. \end{cases}$$

For any 0-1 vector $\boldsymbol{p}$, $\#\boldsymbol{p}$ denotes the number of 1s in the elements of $\boldsymbol{p}$. We denote $\#\boldsymbol{p}_s - \#\boldsymbol{p}_f$ by l. If the given peg solitaire problem is feasible, any feasible sequence consists of l jumps. For example, the peg solitaire problem defined by Figures 1 and 2 is feasible and there exists

a feasible sequence whose length $l = 32$. Since each jump corresponds to an m-dimensional unit vector, a feasible sequence corresponds to a sequence of l unit vectors $(\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^l)$ such that $\boldsymbol{x}^k = (x_1^k, x_2^k, \ldots, x_m^k)^\top$ for all $k \in \{1, 2, \ldots, l\}$ and

$$x_j^k = \begin{cases} 1 & (\text{the } k\text{th move is the jump } j), \\ 0 & (\text{the } k\text{th move is not the jump } j). \end{cases}$$

If a configuration $\boldsymbol{p}'$ is obtained by applying the jump j to a configuration $\boldsymbol{p}$, then $\boldsymbol{p}' = \boldsymbol{p} - A\boldsymbol{u}$ where $\boldsymbol{u}$ is the jth unit vector in $\{0,1\}^m$. From the above discussion, we can formulate the peg solitaire problem as the following integer programming problem;

$$\begin{aligned} \text{IP1: find} \quad & (\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^l) \\ \text{s. t.} \quad & A(\boldsymbol{x}^1 + \boldsymbol{x}^2 + \cdots + \boldsymbol{x}^l) = \boldsymbol{p}_s - \boldsymbol{p}_f, \\ & \mathbf{0} \le \boldsymbol{p}_s - \mathbf{A}(\boldsymbol{x}^1 + \boldsymbol{x}^2 + \cdots + \boldsymbol{x}^k) \le \mathbf{1} \\ & \qquad (\forall k \in \{1, 2, \ldots, l\}), \\ & x_1^k + x_2^k + \cdots + x_m^k = 1 \\ & \qquad (\forall k \in \{1, 2, \ldots, l\}), \\ & \boldsymbol{x}^k \in \{0,1\}^m \\ & \qquad (^\forall k \in \{1, 2, \ldots, l\}). \end{aligned}$$

The problem IP1 has a solution if and only if the given peg solitaire problem is feasible. Clearly, any solution $(\boldsymbol{x}^1, \ldots, \boldsymbol{x}^l)$ of IP1 corresponds to a feasible sequence of jumps.

If we formulate the peg solitaire problem defined by Figures 1 and 2, then the number of variables is $m \times l = 76 \times 32 = 2,432$, the number of equality constraints is $n + l = 32 + 33 = 65$, and the number of inequality constraints is $2 \times n \times l = 2 \times 33 \times 32 = 2,112$. Thus, the size of the integer programming problem is huge and so it is hard to solve the problem by commercial integer programming software.

In Section 5, we propose an algorithm for peg solitaire problem which is a combination of backtrack search and pruning technique based on the above integer programming problem.

3 Linear Relaxation and Pagoda Function

In [3], Berlekamp, Conway and Guy proposed the pagoda function approach for showing the infeasibility of some peg solitaire problems including the well-known problem "sending scout 5 paces out into desert". In this paper, we show that the pagoda function approach is equivalent to the relaxation approach for the integer programming problem.

		−0.3	0.4	0		
		1.0	0	1.0		
0.5	0	0.5	0.4	0.1	0.3	−0.1
0	0.9	0.7	0.3	0.9	1.1	0.4
0.5	0.6	0.1	0.5	0.2	0.6	0.2
		0.8	0	0.8		
		0	0.5	−0.2		

Figure 4: an example of assignment of Pagoda functions

A real valued function pag : $\{1, 2, \ldots, n\} \to \mathrm{R}$ defined on the set of holes is called a *pagoda function* when pag$(\cdot)$ satisfies the properties that for every (vertically or horizontally) consecutive three holes (i_1, i_2, i_3), the pagoda function values $\{\mathrm{pag}(i_1), \mathrm{pag}(i_2), \mathrm{pag}(i_3)\}$ satisfies $\mathrm{pag}(i_1) + \mathrm{pag}(i_2) \ge \mathrm{pag}(i_3)$. (Clearly, the sequence (i_3, i_2, i_1) is also a consecutive three holes, and so the inequality $\mathrm{pag}(i_3) + \mathrm{pag}(i_2) \ge \mathrm{pag}(i_1)$ also holds.) A pagoda function corresponds to an assignment of real values to holes on the board satisfying the above properties. Figure 4 is an example of pagoda function defined on English board.

For any configuration $\boldsymbol{p} \in \{0,1\}^n$, we denote the sum total $\sum_{i=1}^n \mathrm{pag}(i) \times p_i$ by $\mathrm{pag}(\boldsymbol{p})$. The definition of the pagoda functions implies that if a configuration $\boldsymbol{p}'$ is obtained by applying a jump to a configuration $\boldsymbol{p}$, then $\mathrm{pag}(\boldsymbol{p}) \ge \mathrm{pag}(\boldsymbol{p}')$. Thus, if a given peg solitaire problem is feasible, then the inequality $\mathrm{pag}(\boldsymbol{p}_s) \ge \mathrm{pag}(\boldsymbol{p}_f)$ holds for any pagoda function pag$(\cdot)$. So, the existence of a pagoda function pag$(\cdot)$ satisfying $\mathrm{pag}(\boldsymbol{p}_s) < \mathrm{pag}(\boldsymbol{p}_f)$ shows that the given peg solitaire problem is infeasible.

In [7], Kanno showed that there exists a pagoda function which guarantees the infeasibility of the given peg solitaire problem if and only if the optimal value of the following linear programming problem is negative.

$$\begin{array}{lll} \text{PAG-D:} & \min. & (\boldsymbol{p}_s - \boldsymbol{p}_f)^\top \boldsymbol{y} \\ & \text{s. t.} & A^\top \boldsymbol{y} \geq \boldsymbol{0}. \end{array}$$

It is easy to see that for any feasible solution $\boldsymbol{y}$ of PAG-D, the function pag$(\cdot)$ defined by pag$(i) = y_i$ for all $i \in \{1, 2, \ldots, n\}$ is a pagoda function. Thus, it is clear that if the optimal value of PAG-D is negative, then the given peg solitaire problem is infeasible. Unfortunately, the inverse implication does not hold; that is, there exists an infeasible peg solitaire problem instance such that the optimal value of the corresponding linear programming problem (PAG-D) is equal to 0 (see Kanno [7] for example).

The dual of the above linear programming problem is

$$\max\{\boldsymbol{0}^\top \boldsymbol{x} \mid A\boldsymbol{x} = \boldsymbol{p}_s - \boldsymbol{p}_f,\ \boldsymbol{x} \geq \boldsymbol{0}\}.$$

Since the objective function $\boldsymbol{0}^\top \boldsymbol{x}$ is always 0, the above problem is equivalent to the following problem;

$$\begin{array}{lll} \text{PAG-P:} & \text{find} & \boldsymbol{x} \\ & \text{s. t.} & A\boldsymbol{x} = \boldsymbol{p}_s - \boldsymbol{p}_f, \quad \boldsymbol{x} \geq \boldsymbol{0}. \end{array}$$

Thus, there exists a pagoda function which shows the infeasibility of the given peg solitaire problem if and only if the above linear inequality system PAG-P is infeasible.

In the rest of this section, we show that the problem PAG-P is obtained by relaxing the integer programming problem IP1. First, we introduce a new variable $\boldsymbol{x}$ satisfying $\boldsymbol{x} = \boldsymbol{x}^1 + \cdots + \boldsymbol{x}^l$. Next, we relax the first constraint in IP1 by $A\boldsymbol{x} = \boldsymbol{p}_s - \boldsymbol{p}_f$, remove second and third constraints, and relax the last 0-1 constraints of original variables by nonnegativity constraints of artificial variables $\boldsymbol{x}$. Then the problem IP1 is transformed into PAG-P.

We applied pagoda function approach to problems defined on English board and found many infeasible problem instances which do not have any pagoda function showing the infeasibility. Although, the pagoda function approach was a powerful tool for proving the infeasibility of the problem "sending scout 5 paces out into desert", it is not so useful for peg solitaire problems defined on English board.

4 Upper Bound of the Number of Jumps

In this section, we propose a method for finding an upper bound of the number of jumps for each (fixed) jump j contained in a feasible sequence. And additionaly, this method is proved to be a very strong tool to check the feasibilities of the given problems. In the next section, We propose a pruning technique for backtrack search using the upper bound described below.

We consider the following integer programming problem for each jump j;

$$\begin{array}{lll} \text{UB}j\text{:} & \max. & x_j^1 + x_j^2 \cdots + x_j^l \\ & \text{s. t.} & A(\boldsymbol{x}^1 + \cdots + \boldsymbol{x}^l) = \boldsymbol{p}_s - \boldsymbol{p}_f, \\ & & \boldsymbol{0} \leq \boldsymbol{p}_s - \mathbf{A}(\boldsymbol{x}^1 + \cdots + \boldsymbol{x}^k) \leq \boldsymbol{1} \\ & & \qquad (\forall k \in \{1, 2, \ldots, l\}), \\ & & x_1^k + x_2^k + \cdots + x_m^k = 1 \\ & & \qquad (\forall k \in \{1, 2, \ldots, l\}), \\ & & \boldsymbol{x}^k \in \{0, 1\}^m \\ & & \qquad ({}^\forall k \in \{1, 2, \ldots, l\}). \end{array}$$

Since the set of constraints of UBj is equivalent to that of IP1, the given peg solitaire problem is feasible, if and only if UBj has an optimal solution. We denote the optimal value of UBj by z_j^*, if it exists. It is clear that any feasible sequence of the given problem contains the jump j at most z_j^* times. However, the size of the above problem is equivalent to the original problem IP1 and so, it is hard to solve. In the following, we relax the above problem to a well-solvable problem.

To decrease the number of variables, we consider only the pair of starting and finishing configurations and ignore intermediate configurations. The above relaxation corresponds to the replacement of the variables $\boldsymbol{x}^1 + \cdots + \boldsymbol{x}^l$ by $\boldsymbol{x}$. We decrease the number of constraints by dropping second and third constraints of UBj. Then we have the following relaxed problem of UBj;

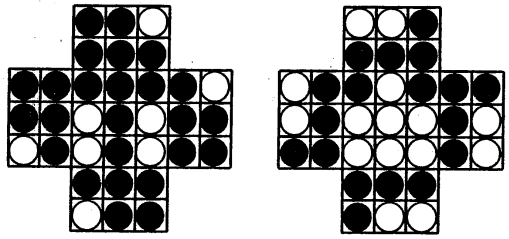

Figure 5: an example of peg solitaire problem

$$\begin{aligned} \text{RUB}j\text{:}\quad & \text{max.} \quad x_j \\ & \text{s. t.} \quad A\boldsymbol{x} = \boldsymbol{p}_s - \boldsymbol{p}_f, \\ & \qquad x_1, x_2, \ldots, x_m \text{ are} \\ & \qquad \text{non-negative integers.} \end{aligned}$$

If a problem RUBj is infeasible for an index $j \in \{1, 2, \ldots, m\}$, the original peg solitaire problem is infeasible and the problem RUBj is also infeasible for each index j. Since the above problem is a relaxed problem of UBj, the optimal value is an upper bound of the optimal value of UBj. If we deal with the problem defined by Figures 1 and 2, the problem RUBj has $m = 76$ integer variables and $n = 33$ equality constraints. The relaxed problems RUBj of the problem defined by Figure 7 are infeasible and so the original problem is also infeasible.

Figure 6 shows the optimal values of RUBj for each jump $j \in \{1, 2, \ldots, m\}$ of the peg solitaire problem defined by Figure 5. Since RUBj is a relaxation of UBj, feasibility of RUBj does not guarantee the feasibility of the given peg solitaire problem. For example, all the relaxed problems RUBj ($j \in \{1, 2, \ldots, m\}$) have optimal solutions as shown in Figure 6 and the given peg solitaire problem defined by Figure 5 is infeasible.

If the problem PAG-P is infeasible, then the problem RUBj is also infeasible for each $j \in \{1, 2, \ldots, m\}$. However, the inverse implication does not hold. For example, PAG-P defined by Figure 7 is feasible, and RUBj is infeasible for each $j \in \{1, 2, ..., m\}$.

Kanno [7] gave 10 infeasible peg solitaire problem instances such that the pagoda function approach failed to show the infeasibility. Our infeasibility check method can show infeasibility of all 10 examples given by Kanno. From the above,

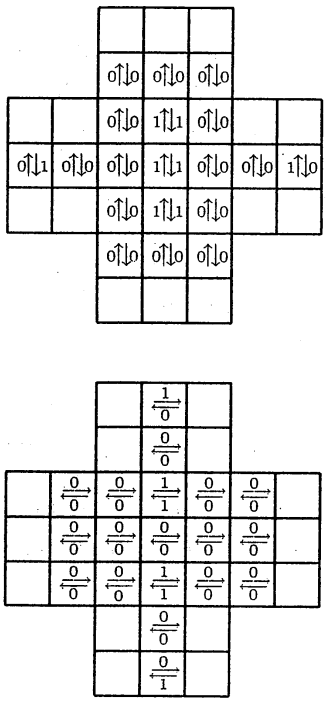

Figure 6: maximal numbers of jumps of Figure 5

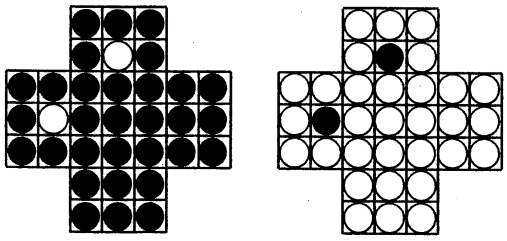

Figure 7: a problem which can be proved to be infeasible

our infeasibility check method compares severely with the pagoda function approach.

Here we note an important comment related to the next section. See Figure 6, we can see that there are many jumps not to be done at all and not to be done twice and so on. So, we can prune many and many branches during the backtrack searching which we deal with in the next section.

Clearly, there are variations of relaxation problems of UBj. We have examined many relaxation problems and chose the above problem RUBj by considering the trade-off between the required computational efforts and the tightness of the obtained upper bound. However, the choice depends on the available softwares and hardwares for solving integer programming problems. In Section 6, we will describe the environment and results of our computational experiences in detail.

5 Backtrack Search

First, we propose forward-only backtrack search algorithm for solving peg solitaire problems. Before executing the backtrack search, we solved the problem RUBj for each jump position $j \in \{1, 2, \ldots, m\}$ and found an upper bound of the number of jumps. We can effectively prune the backtrack search by using the obtained upper bound of the number of jumps. The algorithm is described below:

```
void
solve(configuration start,
      configuration end) {
  table_of_all_jumps  upper_bound;
  int                 rest;

  for (each jump j) {
    solve RUBj;
    if (RUBj is infeasible) {
      print ''infeasible.'';
      exit;
    }
    set the optimal value of RUBj
                  upper_bound[j];
  }

  rest =  the number of pegs of
          the configuration ''start''
        - the number of pegs of
          the configuration ''end'';

  if (search(start, end, upper_bound,
             rest) != true) {
    print ''infeasible.'';
    exit;
  }
}

bool
search(configuration      start,
       configuration      end,
       table_of_all_jumps upper_bound,
       int                rest) {

  if (rest <= 0) {
    if (start == end)
      return true;
    else
      return false;
  }

  for (all possible jumps) {
    if (upper_bound of the jump <= 0)
      continue;

    upper_bound of the jump =
      upper_bound of the jump - 1;
    update the configuration ''start''
      by applying the jump operation.

    if (search(start, end, upper_bound,
               rest - 1) == true) {

      display the configuration
                    ''start'';
      restore the configuration
                    ''start''
        by applying the reverse
                jump operation;
      return true;

    } else {
      upper_bound of the jump =
        upper_bound of the jump + 1;
      restore the configuration
                    ''start''
        by applying the reverse
                jump operation;
    }
```

```
    }

    return false;
  }
```

To avoid searching the same configurations more than twice, we used a hash table with 2,097,169 entries for maintaining all the scanned configurations. If the backtrack search algorithm finds that the present configuration is contained in the hash table, we can stop searching the present configuration. We used the hash table whose size is the maximum prime number currently available at our computer.

Here we point out the importance of the hash technique for solving peg solitaire problem. For example, if we use both IP and hash method, we can solve the problem defined by Figure 8 in 2 seconds on the notebook perspnal computer we will use in the next section. But if we use only IP upper bound or use only hash technique, the program doesn't stop in ten minutes.

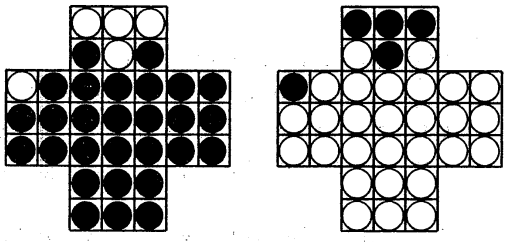

Figure 8: an example problem to which hash is efficient

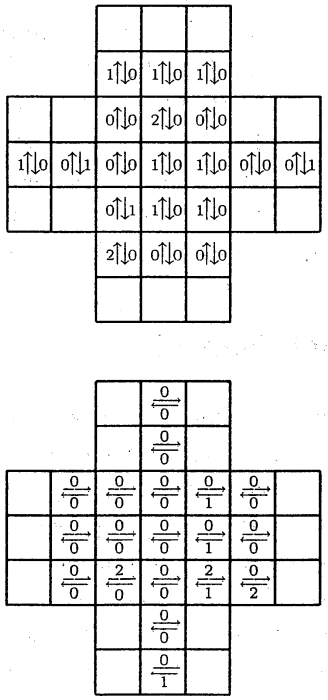

Figure 9: jump upper bounds of the problem defined by Fig. 8

Second, we propose forward-backward backtrack search algorithm. The second algorithm uses the properties of a peg solitaire problem that the number of jumps required to solve the problem is known and solving the problem in forward direction is essentially equivalent to solving the problem in backward direction. Here, solving problem in backward direction means that we start from the finishing configuration, repeat the 'reverse jump' operation and aim to get the starting configuration. Our second algorithm executes backtrack searching from the finishing configuration to the half depth of the search tree and maintains all the obtained configurations by the hash table. Next, the algorithm begins backtrack searching from the starting configuration to the half depth of the search tree. When an obtained configuration is in the hash table, the original solitaire problem is feasible. If all the scanned configurations are not in the hash table, the original problem is infeasible. The idea of the above second algorithm is very simple and effective for some problems but not all. When we applied the second algorithm, some problems require a very large hash table whose size is greater than 2,097,169.

6 Computational Results

In this section, we deal with peg solitaire problems defined on the English board. We used a notebook personal computer with MMX-Pentium 233MHz CPU, 64MB memory, and Linux OS. We solved the relaxed problem RUBj for each j, by the software *lp_solve 3.0*. This *lp_solve* version is released under the LGPL license. One can find the latest version of *lp_solve* at the following ftp site.

ftp://ftp.ics.ele.tue.nl/pub/lp_solve/

In the following, we discuss the problems RUBj ($j \in \{1, 2, \ldots, m\}$), which are called *relaxed problems* in the following. It took about 16 minutes to solve all the 76 relaxed problems defined by Figures 1 and 2. (Since the pair of starting and finishing configurations are symmetrical, we actually need to solve only 12 relaxed problems. However, our computer program does not use the information depending on the symmetricity.) We tried more than 20 peg solitaire problem instances, and the computational time required for solving 76 relaxed problems for each instance is less than 16 minutes. Here we note that there are some problems which took only 10 seconds to solve whole 76 relaxed problems.

We compared the forward-only backtrack searching and the forward-backward search method. The forward-only backtrack search method solves the peg solitaire problem defined by Figures 1 and 2 in 1 second. However, if we apply the forward-backward search method, the hash overflows after 3.5minutes. (Here we note that the program concludes that the hash has overflowed when the 80 percent of the hash is used). Since this problem is symmetric, the forward-only search method finds a feasible sequence easily. However, the symmetricity increases the upperbound of the number of jumps and so the backward search generates many configurations and the hash overflows.

We also tried to solve the symmetrical peg solitaire problem shown by Figure 10. The forward-only search method finds a feasible sequence in 37 minutes. However, the forward-backward search method solves the problem in 1.4 seconds. We think that the performance of two methods depends not only on the symmetricity of the problem but also on the the number of jumps required.

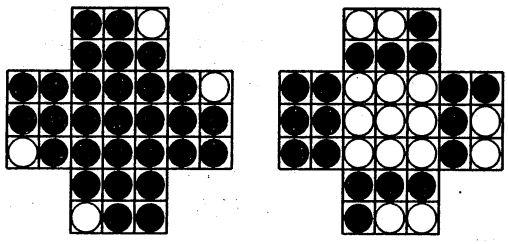

Figure 10: a peg solitaire problem fit for backward and forward searching

The length of the feasible sequence of the problem defined by Figures 1 and 2 is 31 and that of the problem defined by Figure 10 is 13. Since the depth of search tree of the latter problem is small, the hash does not overflow during the algorithm.

We solved more than 20 problems including the instances given in Kanno [7]. Either forward-only search method or forward-backward search method solves each instances we tried in at most 20 minutes. If we select faster algorithm for each instance we solved, the total computational time of our algorithm is less than 20 minutes.

References

[1] Avis, D., and Deza, A.: Solitaire Cones, *Technical Report No. SOCS-96.8*, 1996.

[2] Beasley J. D.: Some notes on Solitaire, *Eureka*, 25 (1962), 13–18.

[3] Berlekamp, E. R., Conway, J. H., and Guy, R. K.: *Winning Ways for Mathematical Plays*. Academic Press, London, 1982.

[4] de Bruijn N. G.: A Solitaire Game and Its Relation to a Finite Field. *Journal of Recreational Mathematics*, 5 (1972), 133–137.

[5] Cross D. C.: Square Solitaire and variations. *Journal of Recreational Mathematics*, 1 (1968), 121–123.

[6] Gardner M.: *Scientific American*, 206 #6(June 1962), 156–166; 214 #2(Feb. 1966), 112–113; 214 #5(May 1966), 127.

[7] Kanno, E.: Linear Programming Algorithm for Peg Solitaire Problems, Bachelor thesis, Department of Mathematical Engineering, Faculty of Engineering, University of Tokyo, 1997 (in Japanese).

[8] Uehara, R., Iwata, S.: Generalized Hi-Q is NP-complete, *Trans. IEICE*, 73 (1990), 270-273.